

# .NET Full Stack Development Program

---

Day 5 I/O & Exception & C# Advance

# Outline

- File Handling (System.IO)
- Exception
- Delegate
- Lambda Expression

# File Handling

# System.IO

IO(Input and Output) is used to process the input and produce the output (read and write data). Most applications need to process some data and produce some output based on the input.

The **System.IO namespace** contains all the classes required for IO operations.

# File class

- C# include **static File class** to perform I/O operations on physical file.
- File class provides functionalities such as create, read/write, delete, etc. for physical files.
- Some methods in the File class:

Method	Description
static FileStream <b>Create</b> (string path)	Create or overwrite a file at the path
static bool <b>Exists</b> (string path)	Determine whether the file exists at the path
static void <b>Move</b> (string sourceFile, string destFile)	Move the source file to the destination
static void <b>Copy</b> (string sourceFile, string destFile)	Copy the source file to the destination
static void <b>Delete</b> (string path)	Delete the file at path

# File class

- Read/Write methods in File class:

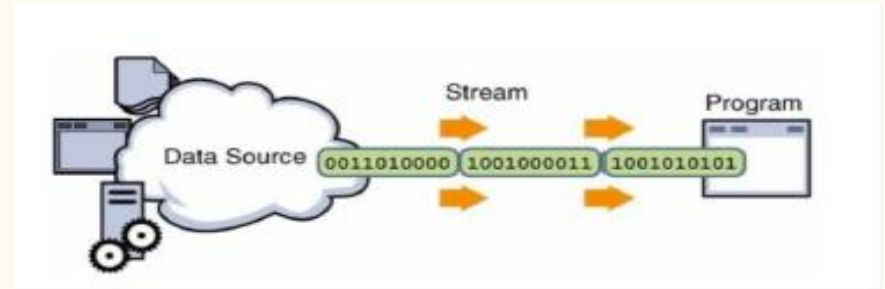
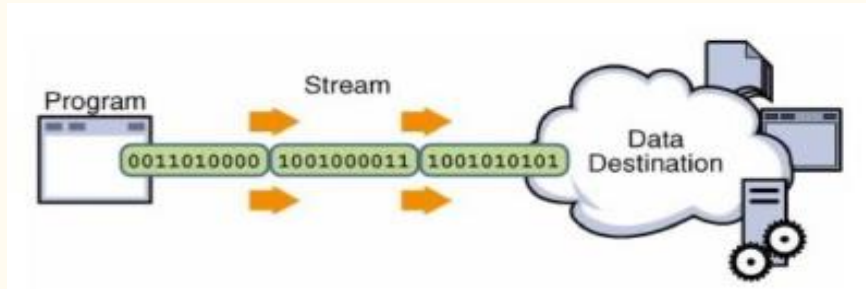
Method	Method
static void <b>WriteAllText</b> (string path, string contents)	Create or overwrite a file, write contents to the file and close the file
static string <b>ReadAllText</b> (string path)	Read from the file and return a string representaion of text

# File Class

- The static File class includes various utility methods to interact with a physical file
- It's used to perform some quick operations on physical file
- It's **NOT** recommended to use File class for multiple operations on multiple files at the same time due to performance reasons, a non-static class would be a better choice in this situation

# Stream

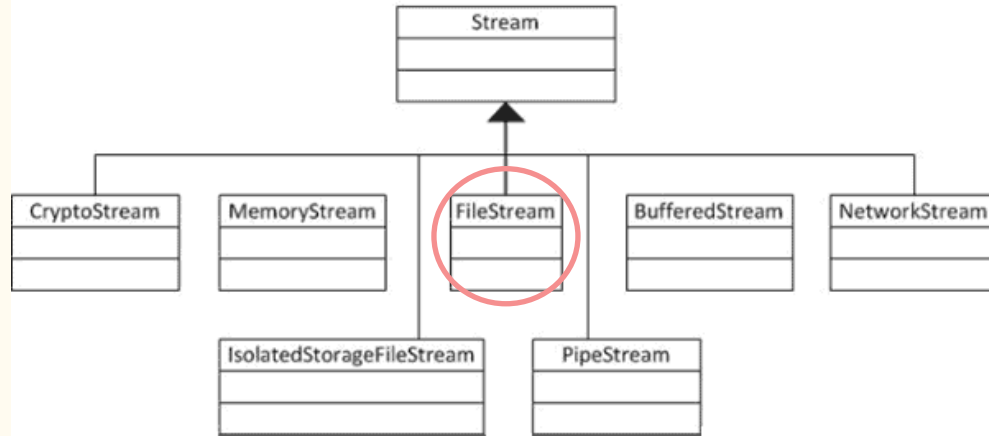
- C# uses the Stream to make IO operation fast.
- A stream is a conceptually endless flow of data. We can either read from a stream or write to a stream.
- A stream is connected to a data source or a data destination.
- It's a good practice to close the stream after use.





# Stream Class

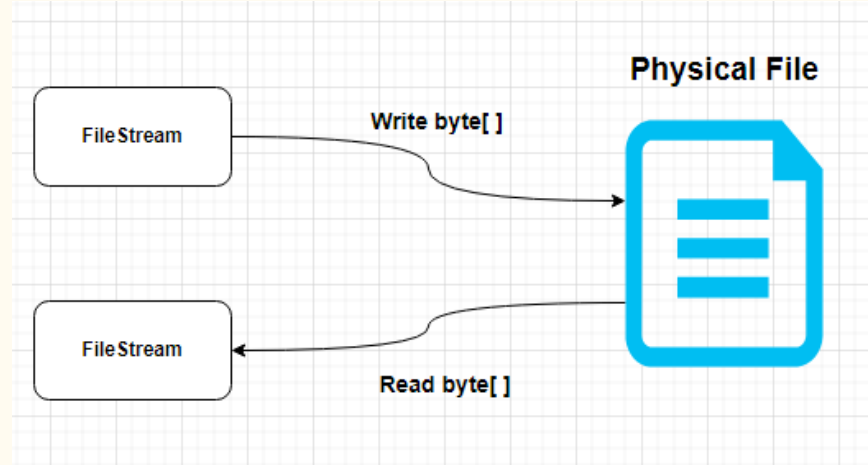
- Stream is an abstract class that provides standard methods to transfer bytes to the source.
- The following classes inherit the Stream class to provide the functionality to Read/Write bytes from a particular source:



Stream Classes Hierarchy

# FileStream Class

- FileStream class provides a stream for file operations. It can be used to perform both **read** and **write operations**.
- To use the FileStream we need to:
  - Include **System.IO** namespace
  - Create an instance of the **FileStream**
  - Perform **read or write operations**
  - Close the stream



# Create an Instance of FileStream

- We need an instance of the FileStream object to create new file or open an existing file.
- There are many constructors available, here are 2 simple ways:

```
//Some Constructors that are often used to create FileStream objects  
public FileStream(string path, FileMode mode);  
public FileStream(string path, FileMode mode, FileAccess access);
```

- **Path:** the path to the file that the current FileStream object will work on;
- **FileMode:** specifies how to deal with the file(create, open, append ...)
- **FileAccess:** determines how the file can be accessed by the FileStream object(read, write...)

# FileMode

FileMode	Description
CreateNew	It specifies that the OS should <b>create a new file</b> . If the file already exists an <i>IOException</i> will be thrown.
Create	It specifies that the OS should <b>create a new file</b> . If the file already exists, it will be <b>overwritten</b> .
Open	It specifies that the OS should <b>open an existing file</b> . If the file doesn't exist, a <i>FileNotFoundException</i> will be thrown.
OpenOrCreate	It specifies that the OS should <b>open an existing file</b> . If the file doesn't exist, a new file will be <b>created</b> . It's useful when reading from the file.
Truncate	It specifies that the OS should <b>open an existing file</b> . When the file is opened, it should be truncated so that its size is zero bytes. Attempts to read from a file opened with FileMode.Truncate causes an <i>ArgumentException</i> exception.
Append	It specifies that the OS should <b>open an existing file</b> , and seek to end of the file, in order to <b>write content at the end</b> . It's useful when writing to the file.

```
namespace System.IO
{
    public enum FileMode
    {
        ...CreateNew = 1,
        ...Create = 2,
        ...Open = 3,
        ...OpenOrCreate = 4,
        ...Truncate = 5,
        ...Append = 6
    }
}
```

# FileAccess

```
namespace System.IO
{
    public enum FileAccess
    {
        Read = 1,
        Write = 2,
        ReadWrite = 3
    }
}
```

FileAccess	Description
Read	It gives read access to the file. Data can be read from the file.
Write	It gives write access to the file. Data can be written to the file.
ReadWrite	It gives read and writes access to the file. Data can be written to and read from the file.

# FileStream

- Perform Read/Write operations:

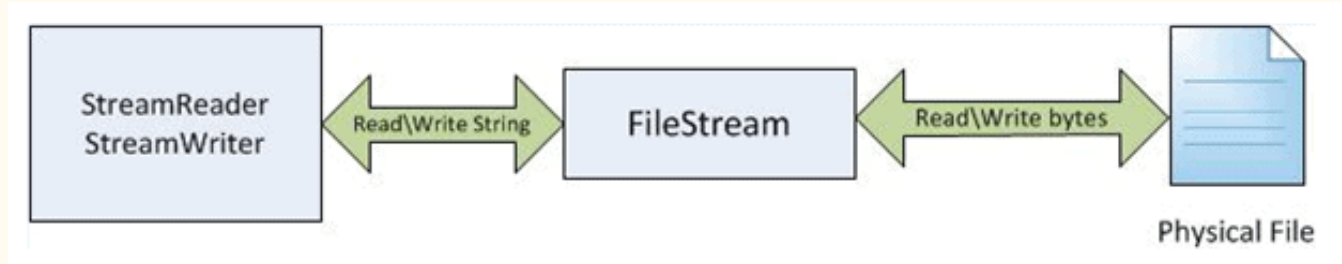
```
// Read() and Write() methods provided by the FileStream  
public override int Read(byte[] array, int offset, int count);  
public override void Write(byte[] array, int offset, int count);
```

- Close the Stream:
  - We can use the **Close()** method to close the stream(the connection to the file).

# StreamWriter and StreamReader

# StreamWriter & StreamReader

- StreamWriter: is a class for writing texts to a Stream by converting characters into bytes.
- StreamReader: is a class for reading characters from a Stream by converting bytes into characters.





# StreamWriter & StreamReader

```
//Some of the constructors in StreamWriter class:  
//initialize the StreamWriter for the specified FileStream  
public StreamWriter(Stream stream);  
//initialize the StreamWriter for the specified file path  
public StreamWriter(string path);  
  
//Some of the constructors in StreamReader class:  
//initialize the StreamReader for the specified FileStream  
public StreamReader(Stream stream);  
//initialize the StreamReader for the specified file path  
public StreamReader(string path);
```

# StreamWriter & StreamReader

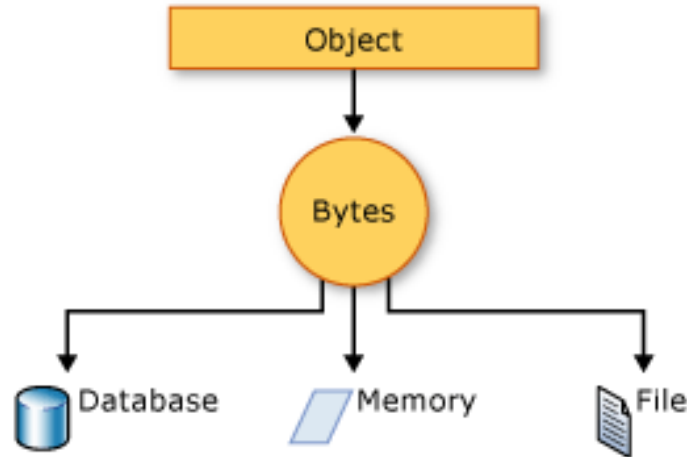
Methods(StreamWriter)	Description
Write(string value)	It writes data to the stream.
WriteLine(string value)	It is the same as Write() but it adds the newline character at the end of the data.
Close()	This method closes the current StreamWriter.

Methods(StreamReader)	Description
ReadLine()	This method <b>reads</b> a line of characters from the current stream and returns the data as a string.
Close()	The Close() method <b>closes</b> the StreamReader object and the underlying stream, and releases any system resources associated with the reader.

# Serialization

# Serialization

- Serialization is **the process of converting an object into a stream of bytes** to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called **deserialization**.



# Exception

# Exception

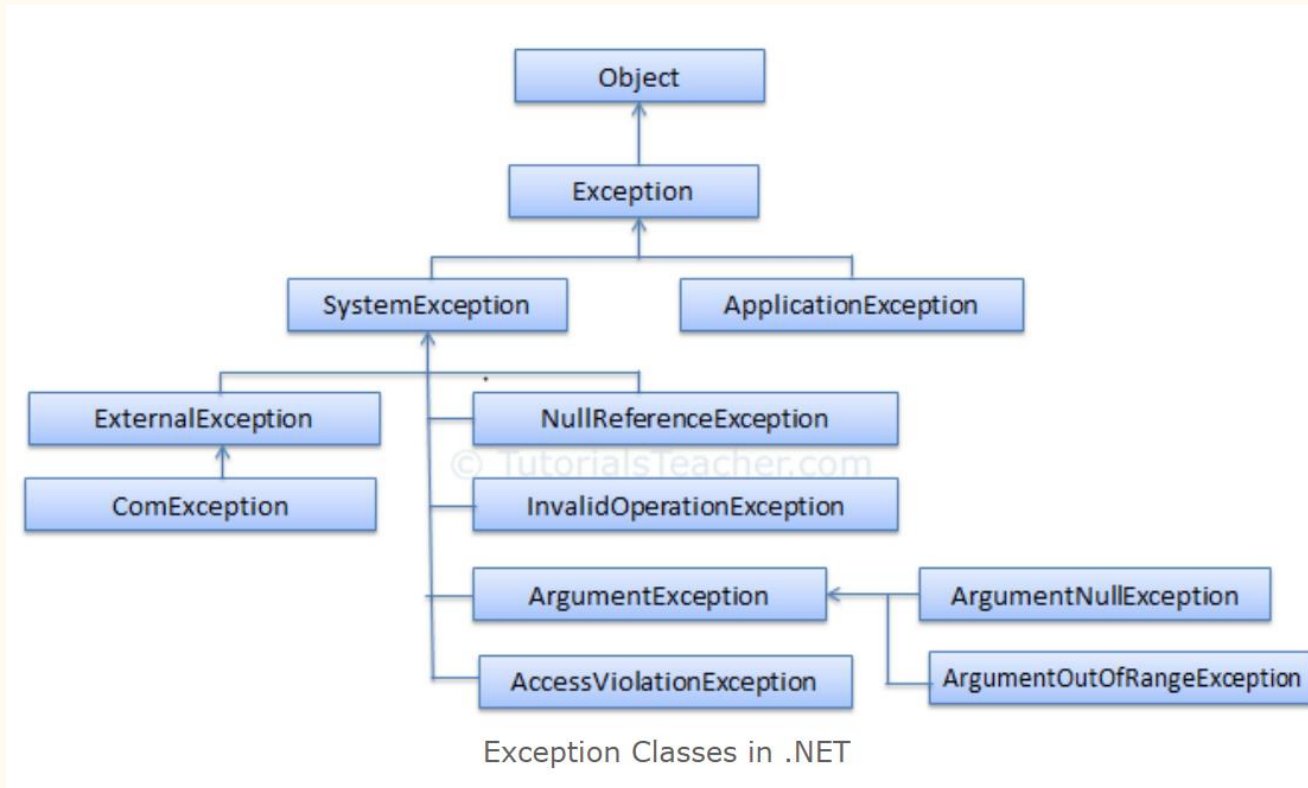
An **exception** is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions. That means the statements placed after the exception-causing statements are not executed but the statements placed before that exception-causing statement are executed by CLR.

**Exception class** in C# is responsible for exceptions

Exceptions can arise due to a number of situations.

- Trying to access the 11th element of an array when the array contains only 10 elements.  
(IndexOutOfRangeException)
- Division by zero (**DivideByZeroException**)
- Accessing a file which is not exist (FileNotFoundException)
- Failure of I/O operations (IOException)

# Exception Hierarchy



# Exception Handling

- try-catch
- finally
- throw



# try-catch

- try/catch block can be placed within **any method** that you feel may raise exceptions
- All the statements to be tried for exceptions are put in a try block
- catch block is used to catch any exception raised from the try block
- If an exception occurs in any statement in the try block, the control immediately passes to the corresponding catch block

# try-catch

- Order of catch is important
  - Always put smaller exceptions in front of bigger exceptions
  - Don't put big basket in the front; you are going to catch everything
  - Don't put small basket in the end; you will not catch anything
- Nested try-catch block is allowed

# finally

- By using a *finally* block:
  - you can **clean up any resources** that are allocated in a try block
  - you can **run code even if an exception occurs** in the try block.
- Typically, ① the statements of a finally block run when control leaves a try statement. ② The transfer of control can occur as a result of normal execution, ③ of execution of a return or break statement, ④ or of propagation of an exception out of the try statement.

2 references

```
internal class Program
```

```
{  
    private StringBuilder sb = new StringBuilder("OgString");
```

```
    //Display the string in the StringBuilder
```

1 reference

```
    private void DisplayString()
```

```
{  
    Console.WriteLine(sb.ToString());  
}
```

```
program.MethodOne();  
program.DisplayString();
```

```
private void MethodOne()
```

```
{  
    try  
    {  
        sb.Append(" + TryOne");  
        MethodTwo();  
    }  
    catch (Exception)  
    {  
        sb.Append(" + CatchOne");  
    }  
}
```

1 reference

```
private void MethodTwo()
```

```
{  
    try  
    {  
        throw new ArithmeticException();  
    }  
    catch (ArithmeticException)  
    {  
        throw new Exception();  
    }  
    finally  
    {  
        sb.Append(" + FinallyTwo");  
    }  
  
    sb.Append(" + Message in MethodTwo");  
}
```

# throw

- Used to explicitly throw an exception
- Useful when we want to throw a user-defined exception.
- The syntax for throw keyword is as follows:
  - throw new ThrowableInstance
    - eg. throw new **ArgumentNullException**();

# throw

```
0 references
static void Main(string[] args)
{
    Student student = null;
    try
    {
        Greeting(student);
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
```

2 references

```
class Student
{
    0 references
    public int Id { get; set; }
    1 reference
    public string Name { get; set; }
}
```

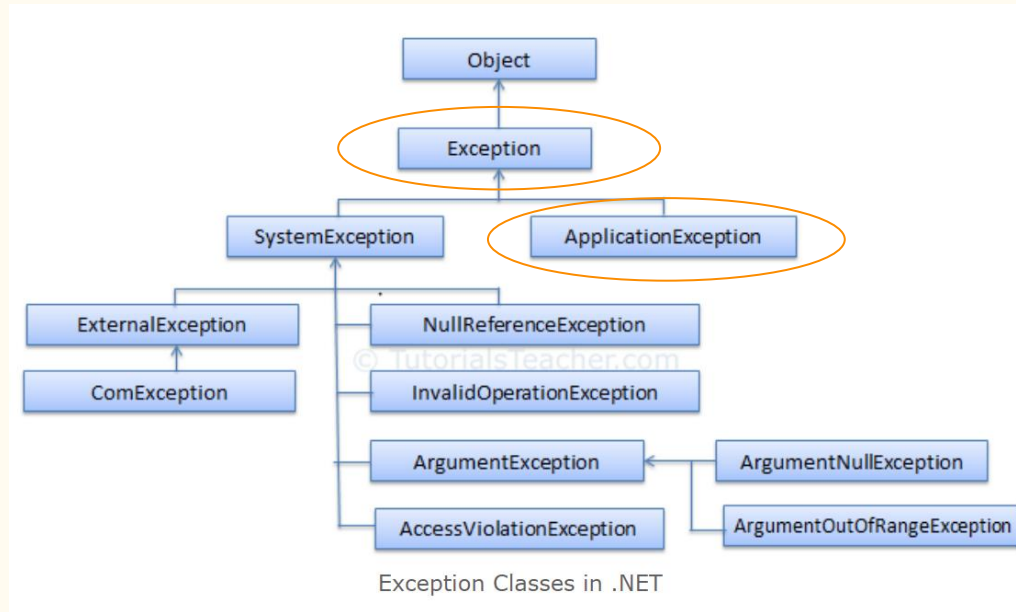
1 reference

```
private static void Greeting(Student s)
{
    if (s == null)
    {
        throw new ArgumentNullException();
    }

    Console.WriteLine("Hello " + s.Name);
}
```

# User-defined Exception

if none of the predefined exceptions meets your needs, you can create your own exception classes by deriving from the **Exception** class



# User-defined Exception

- How to create your Exception:
  - Define a new class inheriting from the **Exception** class
  - Override the virtual members that are defined inside the Exception class based on your need;
  - Throw the custom Exception instance where you need it



# User-filtered Exception Handlers

**User-filtered exception handlers** catch and handle exceptions based on requirements that you defined for the exception. This is useful when one catch statement(a particular exception object) corresponds to multiple exceptions;

use the ***catch*** statement with the ***when*** keyword, only when the condition in `when(condition)` is evaluated as true, the catch statement will get executed.

```
try
{
    //Try statements.
}
catch (Exception ex) when (ex.Message.Contains("404"))
{
    //Catch statements.
}
```

Delegate

# Delegate

- A delegate is a container for holding the reference of a method or function, it's described as a “Function Pointer”.
- A delegate can be declared using the *delegate* keyword, and it can be declared within a class or a namespace(more common).
- The signature of the method must match the signature of the delegate.

```
keyword: delegate  
[access_modifier] [delegate] [return_type] [delegate_name] (params);  
public delegate void MyDelegate(string msg);  
  
//To use delegate  
MyDelegate d;
```

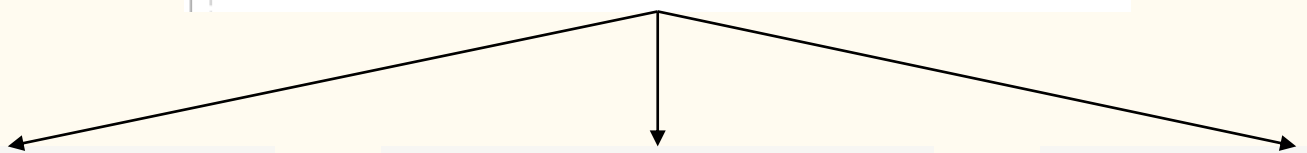
# Delegate

There are 3 steps involved while working with delegates:

1. Declare a delegate (in a class or namespace)
2. Instantiate a delegate
3. Invoke a delegate

# Step1: Declare a Delegate

```
namespace DelegateExamples
{
    //step1: declare
    //declare a delegate with no parameters and no return value
    public delegate void NoParamNoReturnDelegate();
}
```



```
public void MethodA()
{
    Console.WriteLine("Method A");
}
```

```
public void MethodB()
{
    Console.WriteLine("Method B");
}
```

```
public void MethodC()
{
    Console.WriteLine("Method C");
}
```

## Step2: Instantiate a delegate

```
0 references  
static void Main(string[] args)  
{  
    //step 2: set a target method to the delegate  
    NoParamNoReturnDelegate d1 = new NoParamNoReturnDelegate(NoParamNoReturnFunction);  
  
    //another way for step 2  
    NoParamNoReturnDelegate d2 = NoParamNoReturnFunction;  
}
```

```
3 references  
public static void NoParamNoReturnFunction()  
{  
    Console.WriteLine("No params and return void!");  
}
```

## Step3: Invoke a delegate

```
static void Main(string[] args)
{
    //step 2: set a target method to the delegate
    NoParamNoReturnDelegate d1 = new NoParamNoReturnDelegate(NoParamNoReturnFunction);

    //another way for step 2
    NoParamNoReturnDelegate d2 = NoParamNoReturnFunction;
```

```
//step3: invoke a delegate
d1.Invoke();
```

```
//another way for step3
d2();
```

# Multicast Delegate

- The delegate that points to multiple methods is called a multicast delegate.
- The addition operator adds a function to the invocation list, and the subtraction operator removes it.
- If a delegate returns a value, then the last assigned target method's value will be return when a multicast delegate called.

```
1 reference
public class ClassA
{
    1 reference
    public static void MethodA()
    {
        Console.WriteLine("This is A method");
    }
}

1 reference
public class ClassB
{
    1 reference
    public void MethodB()
    {
        Console.WriteLine("This is B method");
    }
}
```

```
namespace MulticastDelegate
{
    public delegate void MyDelegate();
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            MyDelegate md1 = ClassA.MethodA;
            MyDelegate md2 = new ClassB().MethodB;
            MyDelegate md3 = md1 + md2;
            md3();
        }
    }
}
1 reference
```



# Delegate

Delegates are often used as:

- Members of a class
- Parameters of a function/method

# Delegate

- (used as member of the class)

```
namespace MyNamespace
{
    public delegate void MyDelegate(int num);
    public class MyClass
    {
        public int myInt;
        public MyDelegate myDelegate;
        ...
    }
}
```

# Delegate

- (used as parameters of a function/method)

```
public class Exam
```

```
{
```

```
    // Output the result based on the student's score
```

```
    1 reference
```

```
    public void PassExamWithoutDelegate (Student s)
```

```
    {
```

```
        if (s.Score >= 60)
```

```
        {
```

```
            Console.WriteLine("Pass! Score = {0}", s.Score);
```

```
        }
```

```
    else
```

```
    {
```

```
        Console.WriteLine("Fail! Score = {0}", s.Score);
```

```
    }
```

```
}
```

```
// user-defined delegate
```

```
public delegate void PassAsParamExampleDelegate(Student student);
```

```
10 references
```

```
// Output the result based on the student's score using Delegate
```

```
1 reference
```

```
public void PassExamWithDelegate
```

```
    (PassAsParamExampleDelegate d, Student s)
```

```
{
```

```
    d(s);
```

```
}
```

# Build-in Generic Delegates

# Func Delegate

- *Func* is a generic delegate included in the System namespace. It has zero or more input parameters and one out parameter. The last parameter is considered as an out parameter.
- `Func<params_types..., return_type> variableName;`

```
//Func<> -- return a result  
//Func with 2 input parameters and one return  
Func<int, int, int> addNumberDelegate = Sum;  
int result = addNumberDelegate(10, 20);  
Console.WriteLine(result);
```

```
//Func with no input and one return  
Func<bool> returnTrueDelegate = ReturnTrue;  
Console.WriteLine(returnTrueDelegate());
```

1 reference

```
static int Sum(int a , int b)  
{  
    return a + b;  
}
```

1 reference

```
static bool ReturnTrue ()  
{  
    return true;  
}
```

# Action Delegate

- *Action* is a delegate type defined in the System namespace. An Action type delegate is the same as *Func* delegate except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.
- Action<params\_types, ...> variableName;

```
//Action<> -- return void
//Action with 2 input parameters and no return
Action<int, int> printSumDelegate = PrintSum;
printSumDelegate(1, 2);
```

```
1 reference
static void PrintSum(int a, int b)
{
    Console.WriteLine("The result is: " + a + b);
}
```

# Predicate Delegate

- A *predicate* delegate method must take one input parameter and return bool - true or false.

```
//Predicate<> -- take in one parameter return bool  
Predicate<int> isAdultDelegate = IsAdult;  
Console.WriteLine(isAdultDelegate(18));
```

```
1 reference  
static bool IsAdult( int a)  
{  
    return a >= 18 ? true : false;  
}
```

# Lambda Expressions

- Lambda expression in C# is the shorthand for representing anonymous method.
- Syntax:
  - Parameters => Body expression

```
0 references  
public class Test  
{  
    0 references  
    public int AddTwoNumbers(int num1, int num2)  
    {  
        return num1 + num2;  
    }  
}
```

//Delegate without Lambda

```
Test test = new Test();  
Func<int, int, int> addNumber = test.AddTwoNumbers;  
addNumber(2, 3);
```

//Delegate with Lambda

```
Func<int, int, int> addNumberLambda = (x, y) => x + y;  
addNumberLambda(2, 3);
```



# Lambda Expression

- Lambda expression can have zero parameter:

```
() => Console.WriteLine("Hello")
```

- Lambda expression can have multiple parameters in parenthesis ():

```
(a, b, c) => a + b - c
```

- Lambda Expression can have multiple statements in body expression in curly brackets {}

```
a =>
{
    Console.WriteLine("Hello");
    return a + 1;
}
```

- Lambda Expression can be assigned to Func, Action or Predicate delegate

```
Func<Student, bool> isStudentTeenager = s => s.Age > 12 && s.Age < 20;
```

# Outline

- File Handling (System.IO)
- Exception
- Delegate
- Lambda Expression

Question?